An Experimental Evaluation of Specification Techniques for Improving Functional Testing

Kevin L. Mills
National Institute of Standards and Technology
Gaithersburg, Maryland 20899
and
George Mason University[1]
Fairfax, Virginia 22030

kmills@nist.gov
(301) 975-3618

Abstract

Numerous specification techniques have been proposed to improve the development of computer software. The present paper reports results from one experiment to compare the effectiveness of functional testing given three different sets of software specifications. Three groups of experienced programmers independently developed functional test cases for the same automated cruise control software. One group worked solely from a natural language description, one was given a graphical, real-time, structured analysis (RTSA) specification to augment the natural-language description, and a third was given, in addition to the English-language description and the RTSA specification, an executable specification. Two measures of performance were evaluated: 1) the degree of statement coverage achieved by the test cases created by each group and 2) the amount of time taken to create the test cases. No significant difference in performance was found among any of the groups for either of the measures. Individual differences in performance among the subjects were observed. Additional statistical tests were used to evaluate the effect of other factors on these individual differences. Such differences appear to be attributable to variations among the subjects in effort, ability, motivation, and understanding of the assignment. These results suggest that software engineering researchers should investigate people-related, management issues that, perhaps, provide the most significant, short-run influence on the performance of software engineers.

[1] The author attends George Mason University as a doctoral student while employed by the National Institute of Standards and Technology. The work described in this paper was conducted during a doctoral seminar at George Mason.

KEYWORDS: EXECUTABLE SPECIFICATIONS; FUNCTIONAL TESTING; RTSA; SPECIFICATION TECHNIQUES

1.0 Introduction

Numerous specification techniques have been proposed to improve the development of computer software [Bailin 1989, Diller 1990, Harel et al 1987, Heninger 1980, ISO 1987, ISO 1989, Ward and Mellor 1985-1986]. Such techniques aim to remedy the shortcomings of natural language, that is, to increase precision and to identify omissions and inconsistencies in specifications. Some specification techniques go further by providing an underlying semantic model that can be executed to explore the operational properties of a proposed software system [Harel et al 1990, Lee and Sluzier 1991, Zave and Schell 1986]. Occasionally, experiments have been conducted to evaluate the effectiveness of various specification techniques at improving the design and implementation of software [Basili et al 1986, Kelly and Murphy 1990]; however, few experiments have been conducted to evaluate the effect of specification techniques on functional testing. As used in this paper, functional (or black-box) testing refers to testing a software system against a specification of the functions that the software is to perform, rather than testing against the code (white-box testing) that implements those functions. Functional tests are written without the tester examining the code, while white-box tests are written by the tester with full knowledge of the code. The present paper reports results from one experiment to compare the effectiveness of functional testing given three different sets of software specifications.

In the first case, an English-language description alone provides the information used to create functional test cases. In a second case, the English-language description is augmented with a graphical specification notation known as real-time structured analysis (RTSA) [Ward and Mellor 1985-1986]. In the third case, the English-language description and the RTSA specification are used, along with an executable specification. The experiment investigates whether these forms of specification affect the ability of experienced programmers to create functional tests for a given application. Two working hypotheses are adopted. First, *the use of RTSA will improve the test coverage and decrease the time taken to create functional tests* over that which is achieved when only the English-language description is available. The introduction of a graphical notation, such as RTSA, should increase the precision of the specification, allowing test developers to more readily identify the test cases needed to cover software written to meet the specification. In addition, obtaining a given coverage should take less time with the RTSA specification because those using only the natural-language description would need to translate the natural language into a more precise model of the requirements in order to write effective test cases. No generally accepted measure exists for the effectiveness of functional testing. In this experiment, statement coverage is used to assess the effectiveness of functional testing. Although statement coverage is a measure typically associated with white-box testing, the software used in this experiment might allow statement coverage to provide a useful measure for the effectiveness of functional testing because most of the code composing the application software maps directly from the natural-langauge description. Eleven of the eighteen modules in the software correspond directly to paragraphs in the natural-langauge description. Four of the remaining seven modules provide control for sequencing application modules, while the final three modules provide encapsulate data elements taken from the natural-language description.

A second working hypothesis is that *the use of an executable specification will improve the test coverage over that which is achieved with a natural-language description and a RTSA specification.* The marginal improvement gained with the executable specification is expected to stem from the capability to dynamically model the problem, a capability that should reduce misunderstandings in some cases. In addition, since the executable specification can interpret the functional test language, the intent of each test case can be verified prior to using the test case against the actual application software. Use of an executable specification should increase the time taken to write tests over the time taken when only a natural-language description and a RTSA specification are available. This extra time will be used to dynamically evaluate the functional tests.

The remainder of the paper describes an experiment intended to test these hypotheses. The experiment design is presented, followed by the details of conducting the experiment. Then, the results from the experiment are reported. Finally, conclusions are drawn. Due to length limitations, much of the supporting experimental material can only be outlined here. The complete set of materials, packaged as an experiment notebook, can be obtained from the author [Mills 1993].

2.0 Experiment Design

The overall design of the experiment required replicated development of functional tests by three groups of experienced, professional programmers. Figure 1 provides a pictorial overview of the general design. The experiment subjects were drawn from a population of about 200 experienced programmers working within the Computer Systems Laboratory at the National Institute of Standards and Technology (NIST). In Figure 1, this population is represented as a grid with the black boxes denoting a sample drawn from the population. The author approached potential subjects with a memorandum [Mills 1993, Tab A] that explained the purpose of the experiment and solicited volunteers. The experiment called for three groups of at least five test

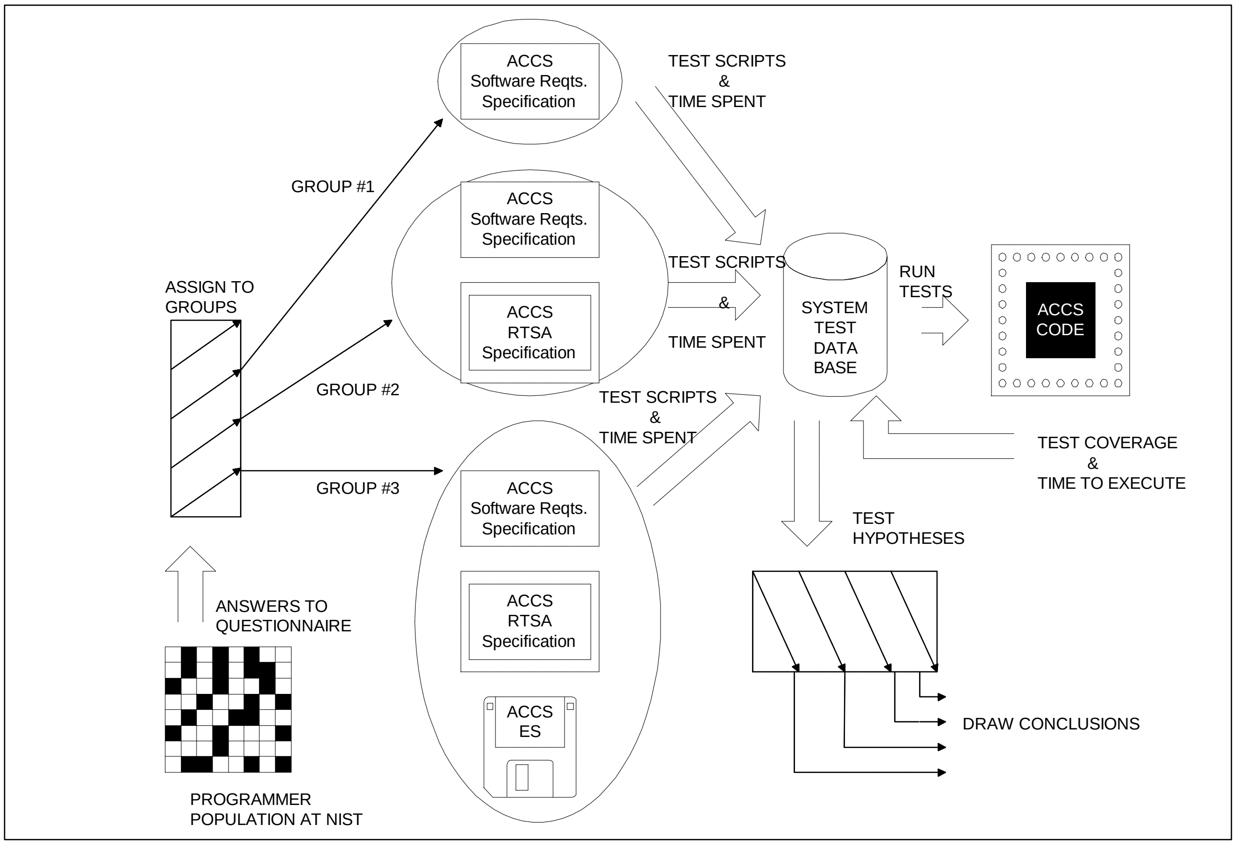
ACCS Software Reqts. Specification
TEST SCRIPTS & TIME SPENT
GROUP #1
ACCS Software Reqts. Specification
ACCS RTSA Specification
TEST SCRIPTS & TIME SPENT
ASSIGN TO GROUPS
SYSTEM TEST DATA BASE
RUN TESTS
ACCS CODE
GROUP #2
TEST SCRIPTS & TIME SPENT
TEST COVERAGE & TIME TO EXECUTE
GROUP #3
ACCS Software Reqts. Specification
TEST HYPOTHESES
ANSWERS TO QUESTIONNAIRE
ACCS RTSA Specification
ACCS ES
DRAW CONCLUSIONS
PROGRAMMER POPULATION AT NIST

Figure 1. Overview of the Experiment Design

writers. Each group was initially staffed with eight subjects in order to account for the possibility that some subjects might drop out of the experiment. Each subject was asked to respond to a questionnaire [Mills 1993, Tab B] requesting a name, an electronic mail address, the number of years of programming and functional testing experience, and the degree of familiarity with software requirements specifications, RTSA, and executable specifications. Subjects were assigned to groups on the basis of their answers to the questionnaire and on the author's assessment of their abilities. The author had once supervised or managed nineteen of the twenty-four subjects and had worked with the remaining five as colleagues. These relationships occurred over the past five years. In these relationships, the author had ample opportunities to gain an appreciation for the strengths and weaknesses of the subjects.

After comparable groups were formed, each group was assigned randomly to receive different sets of specifications for an automated cruise control system (ACCS). The ACCS application was excerpted from a real-time application suggested by Gomaa [Gomaa 1993]. Gomaa's application includes both cruise control functions and vehicle monitoring functions. To reduce the scope of the application, while retaining an interesting level of complexity, the author selected the cruise control subsystem from Gomaa's application and then used that subsystem as the application for the experiment.

As a first step in describing the ACCS, the author excerpted relevant details from Gomaa's RTSA specification and revised them to fit the context of the experiment. A context diagram for the ACCS is shown in Figure 2. The application consists of four input devices and a single output to a throttle positioner. The ACCS is decomposed into two additional levels of data/control flow diagrams as shown in Figures 3 and 4. The control transform, Cruise Control, shown in Figure 4, is further described with a state transition diagram as rendered in Figure 5. The ACCS must process nine events in six different states. From these diagrams, the ACCS RTSA specification was completed with a data dictionary and mini-specifications for each of the data transforms. The complete RTSA specification for the ACCS is contained in the experiment notebook [Mills 1993, Tab H].

Once the RTSA specification was completed, a seven-page, natural-language description for the ACCS was drafted. This description details the ACCS hardware (including a block diagram of the system, a description of the cruise control lever, and a specification of all register formats), presents four typical user scenarios, and delineates the functional requirements of the ACCS (as derived from the RTSA specification). The complete English-language description is available in

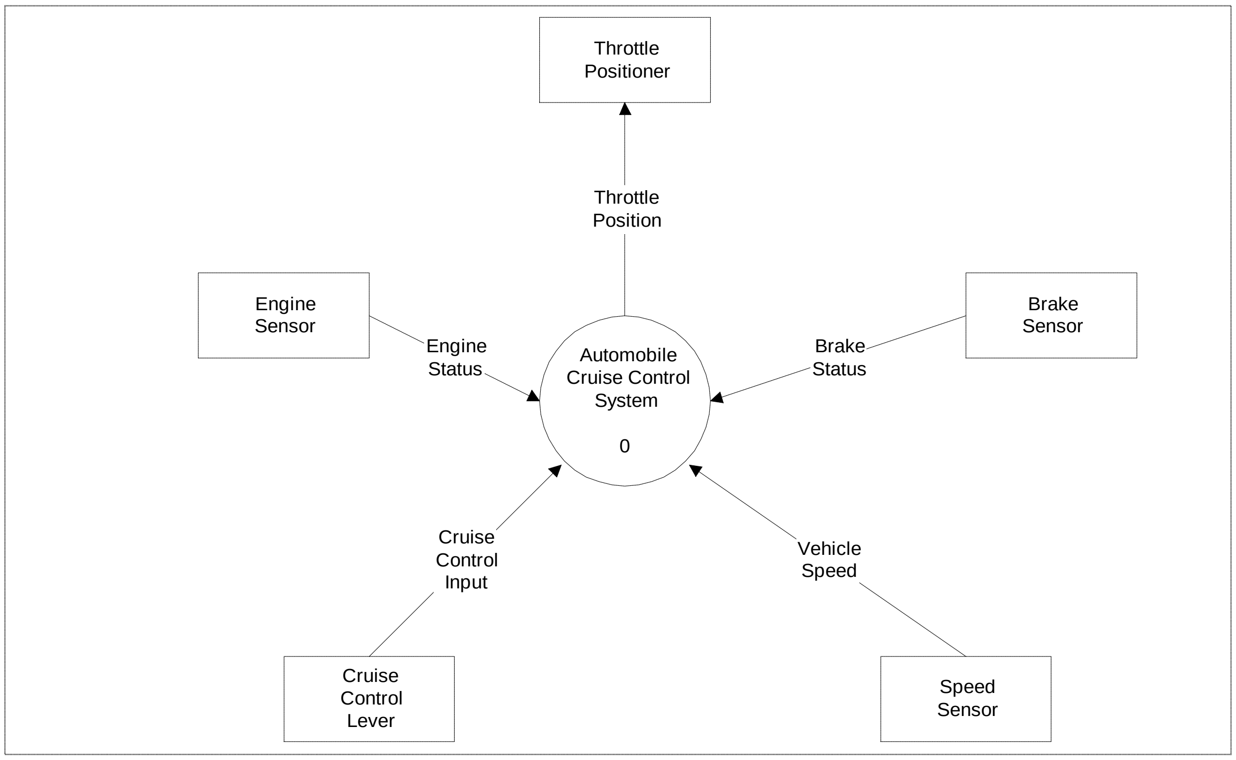

Figure 2. Automobile Cruise Control System Context Diagram

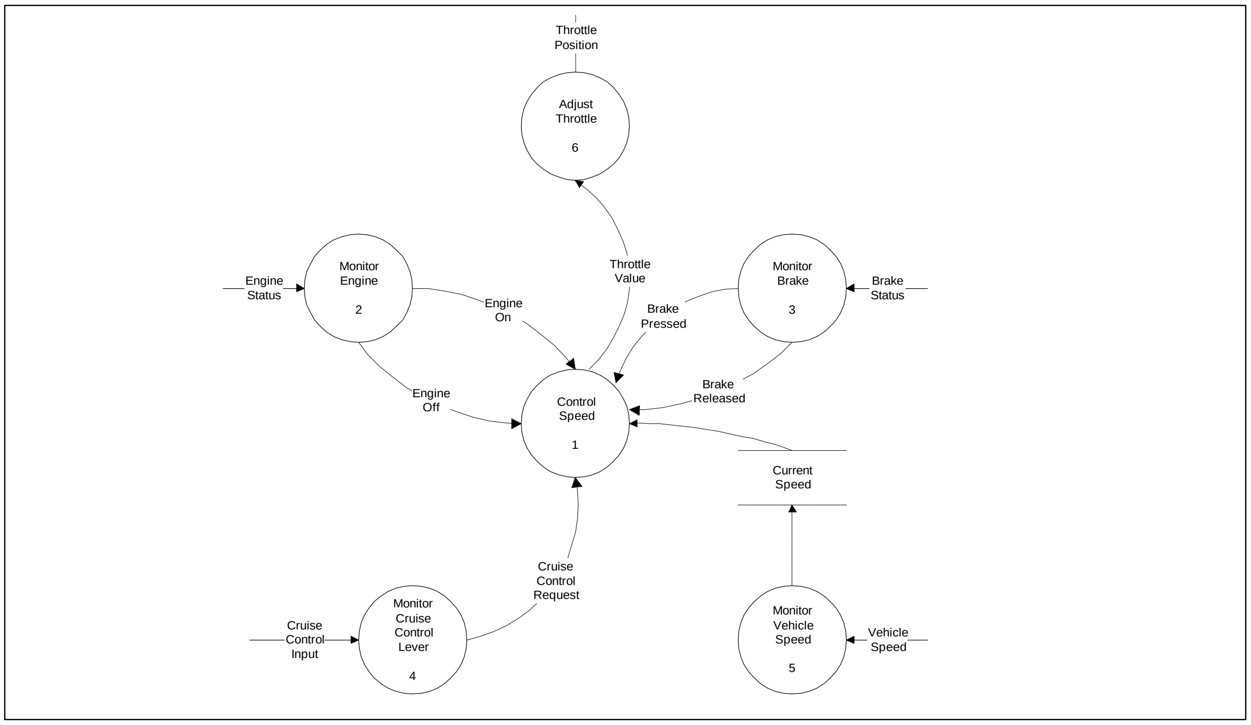

Figure 3. First Level Decomposition Of ACCS

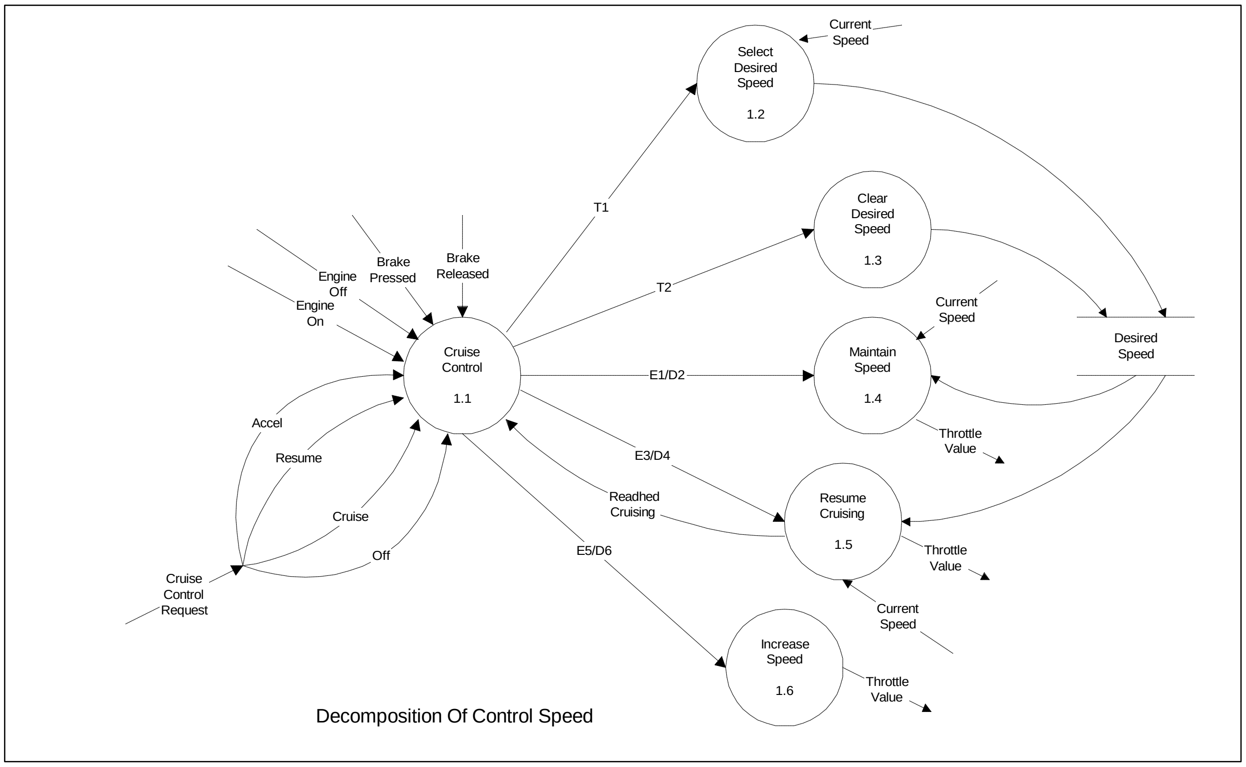

Figure 4. Decomposition Of Control Speed

the experiment notebook [Mills 1993, Tab F].

A third specification of the ACCS was developed using the CLIPS expert systems language (version 6.0) [NASA 1993]. The CLIPS specification encodes the RTSA specification as a set of executable rules. The CLIPS source code for the executable specification is included in the experiment notebook, [Mills 1993, Tab J]. The CLIPS specification also includes rules for interpreting a functional, test language designed by the author for creating test cases for the ACCS.

The ACCS test language consists of two sets of test directives: 1) test management and 2) automobile control. The test-management directives include **begin**, **end**, and **comment**. Begin and end directives bracket each test case and comments are used to explain the intent of a test.

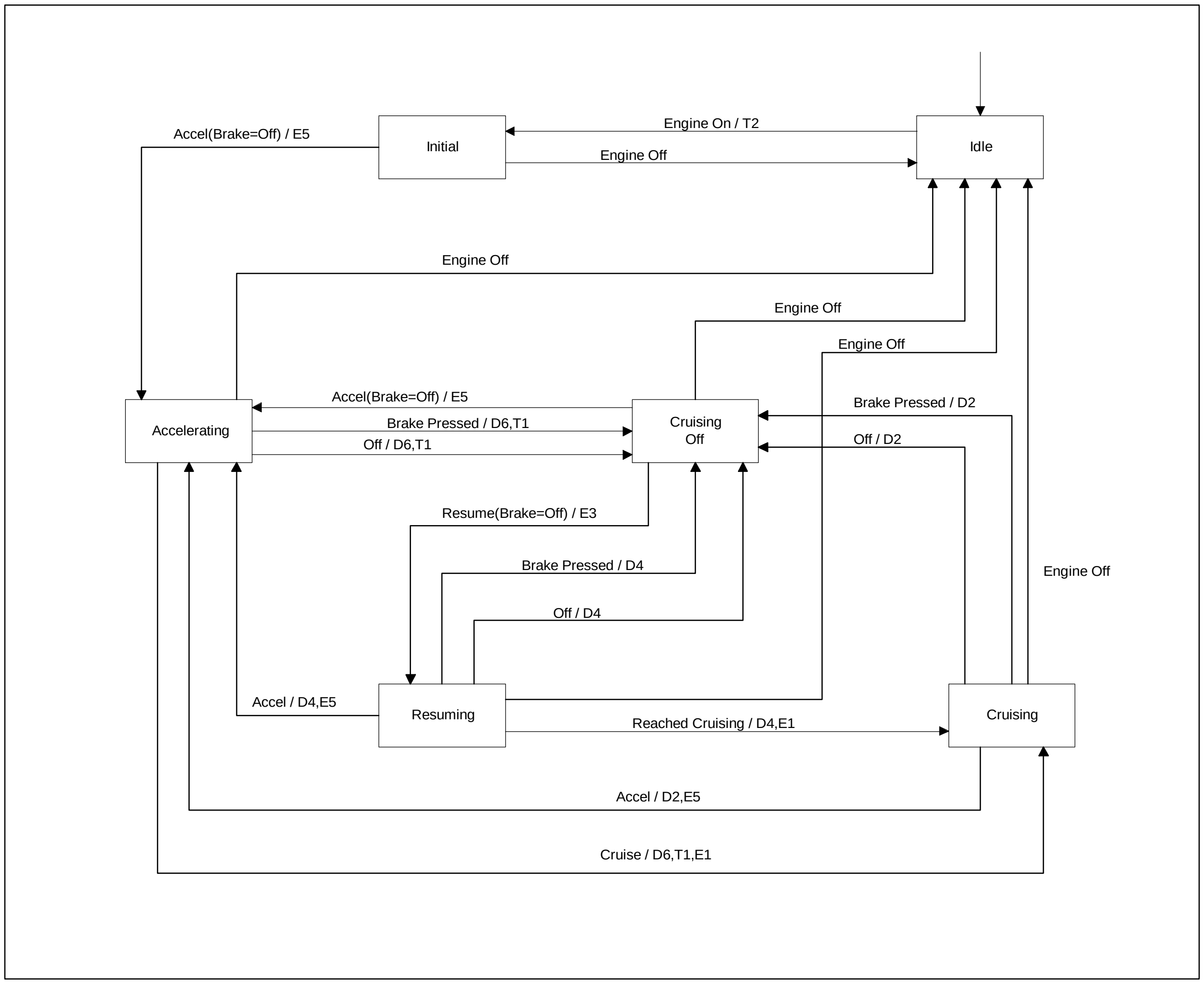

Figure 5. State Transition Diagram for Cruise Control

The automobile-control directives include: 1) **speed**, 2) **engine**, 3) **brake**, and 4) **lever**. Speed is used to simulate changes in vehicle speed. Engine and brake are used to simulate actions taken by the driver to ignite and extinguish the engine, or to depress and release the brake. Lever allows simulation of a driver adjusting the cruise control lever. Some automobile control directives enable a tester to indicate that a driver takes an action at a particular vehicle speed. Specifically, the brake could be depressed, the cruise control lever could be set to the OFF position, or the cruise control lever could be released from the ACCEL position at specific vehicle speeds. As will be explained below, the test language should also have allowed other directives to be triggered at specific speeds. A complete specification of the syntax and semantics of the test language (see the experiment notebook [Mills 1993, Tab G]) was distributed to each subject. Also distributed were four sample test scripts, one corresponding to each of the user scenarios given in the English-language description. The test language was used by the subjects to create functional test scripts for the ACCS.

The author chose to conduct the experiment using working conditions rather than laboratory conditions. This meant that the tasks were assigned, the subjects were given duration deadlines, and the subjects were to track their own hours. The hours of effort recorded by the subjects themselves are potentially inaccurate; however, closely monitoring the time spent by each subject would violate the intent to carry on the experiment under working conditions. Each subject was to return test scripts, along with an estimate of the amount of time spent reviewing the ACCS specification, thinking, and writing test cases, to the author via electronic mail. Each set of test scripts was reviewed for correct syntax and then executed against an implementation of the ACCS.

The ACCS implementation was written by the author in C and executed on a SUN SPARC 2. The implementation was augmented by a test language driver and a cruise control hardware simulator, as shown in Figure 6. Source code for all three components of the system is included in the experiment notebook [Mills 1993, Tabs L, M, and N]. Tests can be executed interactively or from disk files, and test outputs can be written to a screen or to a log file. The solid lines in Figure 6 represent data exchange and the dashed lines represent function calls. The Test Language Interpreter changes the state of the simulated hardware as directed by test cases and then calls the ACCS implementation. The ACCS implementation extracts hardware state from the simulator and executes as necessary to respond to changes in hardware state.

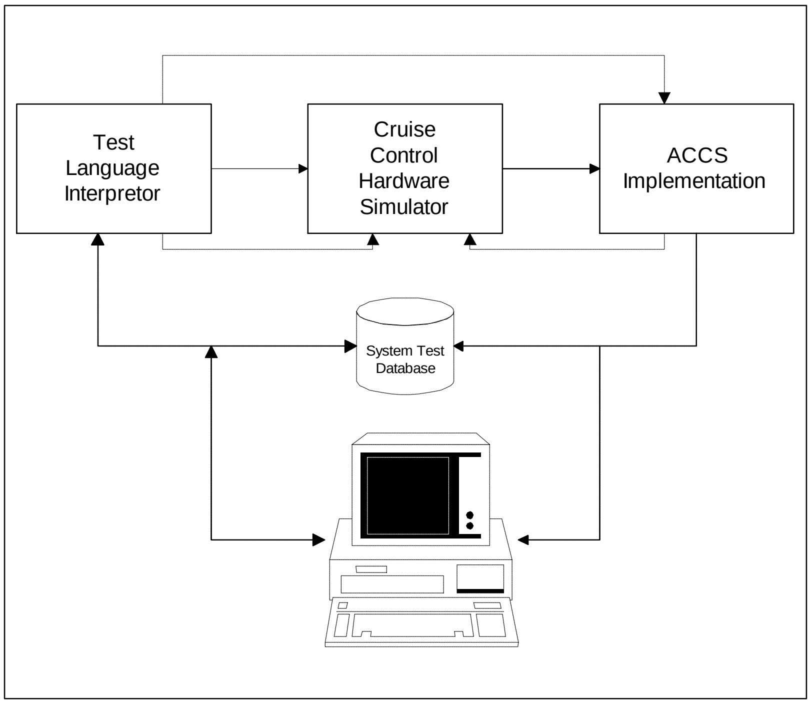

Figure 6. ACCS Test System Architecture

The ACCS implementation consists of 508 lines (without comments) of code comprising eighteen modules. Together, these modules contain 116 basic blocks (a basic block is a sequence of code that, once entered, executes completely as a unit). The software design for the ACCS implementation is shown as a structure chart in Figure 7. Each rectangle represents a module of the ACCS; included inside the rectangle is the module name and the number of basic blocks (bbs) contained within the module. The largest module, process events (43 bbs), encapsulates the cruise control state transition diagram. Seven basic blocks out of the 116 total could not be covered. This means that the best achievable coverage is 109/116, or 93.97%. Three of the seven uncoverable basic blocks handle default situations that could not occur under normal circumstances, for example, if the ACCS reaches an invalid state or, for a given state, if an invalid event occurs. The remaining four uncoverable blocks result from flaws in the test language. For example, the engine could not be turned off in every state even though ACCS code was written to handle such an occurrence. Since the test driver code was

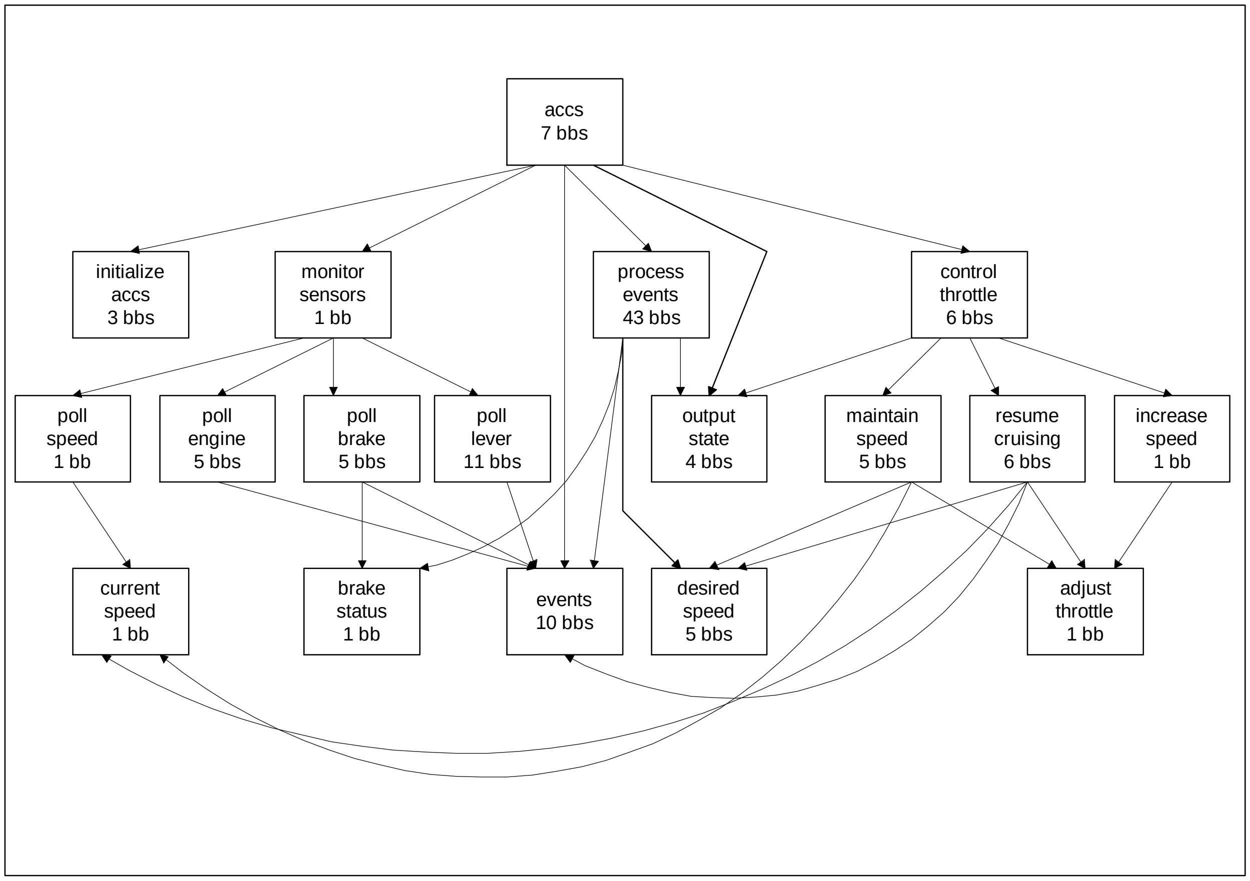

Figure 7. Software Design Of The ACCS Implementation

sequential, any events that required asynchronous occurrence had to be triggered by some future ACCS state (usually, vehicle speed). For most asynchronous events, the test language included a means to specify under what conditions the event should be triggered. Unfortunately, the author forgot to include, in the test language, the capability to trigger asynchronously the ENGINE OFF event.

When the ACCS is executed without any work to do, 21 basic blocks are covered (this amounts to 18.1% statement coverage). When the ACCS is stimulated with the four sample test cases included with the test-language specification, 103 basic blocks are covered. Any subject who simply returned the sample test cases would achieve a statement coverage of 103/116, or

88.79%. The fact that each participant received sample test cases that achieved 89% test coverage tends to bias the experiment against the hypotheses by limiting the room for achieving improved test coverage to only 5% (assuming that each participant begins with the sample test cases and then adds to them); however, a review of the test cases received from the participants revealed that only a few based their work on the sample test cases. In fact, those participants who achieved the best test coverage did not base their work on the sample test cases. This observation is further corroborated by the fact that several of the participants achieved test coverage measures below that which would have been achieved simply be returning the sample test cases. The author suspects that these flaws, which artificially compress the number of basic blocks available to differentiate performance among individuals, might decrease the variability of the results, and thus bias the experiment against the hypotheses.

The ACCS, as written, outputs significant application events to a log file, but collects no coverage or execution time statistics. These data collection requirements were satisfied with existing Unix programs. The ACCS was compiled using the AT&T C compiler **-a** option. This option: 1) generates an **accs.d** file where coverage data can be accumulated and 2) inserts into the ACCS object code calls to coverage-counting routines. As the ACCS is executed, coverage data is collected and then accumulated to the **accs.d** file at the close of each execution. A separate program, **tcov**, converts the **accs.d** file into an annotated source listing of the ACCS that includes coverage statistics. To satisfy the requirement for execution time recording, each test execution of the ACCS was run as a child of the Unix command **time**. The **time** command prints the amount of wall-clock and processor time used by its child.

3.0 Experiment Conduct

The experiment was initiated when the author approached 28 potential subjects with a memorandum explaining the purpose and scope of the experiment, and outlining the work that would be required of each subject. Twenty-four of the 28 potential subjects volunteered to participate and filled in a brief questionnaire. After analyzing responses to the questionnaire, the author determined the composition of an idealized group from among the volunteers. The subjects were then allocated into three groups, while attempting to approach the ideal in each group. Table 1 shows the results from this process. The traits were allocated to individuals in such a manner that an ideal grouping could not be achieved. After assigning subjects to groups, each group was allocated randomly to one of the three classes called for in the experiment design.

Experiment materials, along with a covering memorandum [Mills 1993, Tab A], were distributed to each subject. Subjects in all groups were given the natural-language description for the ACCS and an appendix specifying the syntax and semantics of the test language. Subjects in groups two and three were also given an appendix containing an RTSA specification for the ACCS (as well as a brief description of RTSA notation). Subjects in group three received a diskette containing an executable version of the ACCS specification (as well as an appendix indicating how to load and operate the executable specification). No training was given to the subjects.

Each subject was asked to develop test scripts that will "...thoroughly test the ACCS software against the functional requirements [Mills 1993, Tab A] ." A more formal definition of the task might have been given, for example indicating that the measure of performance in the experiment was statement coverage. The author chose to give the type of vague guidance that most functional testers receive, rather than give the more formal guidance that might be expected under laboratory conditions.A desired deadline of three weeks hence was set for finishing the task.

Trait	Value	Ideal Group	Group One	Group Two	Group Three
Programming Experience	>15 years	2	2	2	2
	10-15 years	3	3	2	3
	5-10 years	3	3	4	3
Functional Testing Experience	>9 years	2	3	1	1
	1-9 years	4	3	5	4
	None	2	2	2	3
Software Reqts. Specification Experience	R/W	5	3	7	4
	R/NW	2	4	1	3
	NR/NW	1	1	0	1
RTSA Experience	R/W	2	2	2	1
	R/NW	1	1	0	2
	NR/NW	5	5	6	5
Executable Specification Experience	U/W	4	4	3	4
	U/NW	1	1	1	1
	NU/NW	3	3	4	3
Cruise Control Experience	Used	7	7	7	7
	Used 0-1	1	1	1	1

Those who could not meet the deadline were asked to complete the work as soon as possible thereafter. The test scripts were to be submitted to the author using electronic mail. Three mailing lists, one for each group, were established so that pertinent information, regarding corrections to or clarifications of the experiment materials, could be readily distributed. The mailing lists were used to broadcast corrections or clarifications to documents. Inquiries concerning other issues were not circulated on the mailing lists.

Soon after the materials were distributed, one of the subjects in group two withdrew from the experiment by declaring that the amount of work involved was more than anticipated. After two weeks, another subject, this time in group three, withdrew from the experiment after indicating a lack of free time for writing scripts to test cruise control software. As the initial deadline passed, only two subjects had submitted test cases, but within another week, thirteen subjects (seven in group one, four in group two, and two in group three) had returned test cases. Over the course of the experiment (about eight weeks), 20 of the 24 subjects returned test scripts, and four dropped out (two each from groups two and three).

Each subject developed the functional test scripts without knowledge of the ACCS software. None of the test scripts were run against the ACCS software prior to submission. (Recall, though, that the subjects in group three were given an executable specification against which to run their tests.)

As test cases were received, they were reviewed and executed against the ACCS implementation. The data collected is shown in Tables 2, 3, and 4.

4.0 Experiment Results

Some of the data collected appears rather surprising and warrants investigation. For example, since each subject was given four sample test scripts (containing 86 test directives, of which 33 were comments) that, taken together, achieved coverage of 103 basic blocks (88.79%),a reasonable expectation was that every subject would at least equal this score by beginning with the sample tests and adding tests in an attempt to achieve increased thoroughness. Yet, in each

Table 2. Data Collected For Group One

(Natural-language Specification)

Subject	Test Statements total (comments)	Creation Time	Execution Time real (cpu)	Coverage percent (bbs)	Date Received
1	110 (46)	5 h	0.3 s (0.1s)	90.52 (105)	10/24
2	85,955 (17,229)	4 h	158.8 s (58 s)	87.07 (101)	10/24
3	124 (49)	4 h	0.3 s (0.2 s)	88.79 (103)	10/24
4	38,326 (8,208)	5 h	72.6 s (24.1 s)	87.93 (102)	10/14
5	79 (32)	2.75 h	0.3 s (0.1 s)	77.59 (90)	10/25
6	319 (80)	8 h	0.5 s (0.2 s)	89.66 (104)	10/24
7	197 (47)	5 h	0.3 s (0.2 s)	89.66 (104)	10/22
8	145 (22)	5 h	0.3 s (0.1 s)	89.66 (104)	10/24

group, some subjects failed to cover at least 103 basic blocks. An examination of each of these cases is enlightening.

In group one (natural-language description only), three subjects (2, 4, and 5) failed to cover at least 103 basic blocks. Subjects 2 and 4 used programs to automatically generate test cases. One of the subjects even submitted the generator along with the test cases. This approach proved relatively ineffective. A massive number of test statements were generated in both cases, and a substantial amount of time was consumed executing the tests, yet only 101 and 102 basic blocks were covered. In addition, no time savings were realized over those who generated test cases by hand. Subject 5 represents a different case. Here, only 2 and 3/4 hours were spent reviewing the material and creating test cases. Only 47 test directives (after removing comments) were created. These facts, together with the lateness of the submission, suggest that the subject was just trying to fulfill a commitment with the minimum possible effort. The results, 90 basic blocks covered, support this conclusion.

Table 3. Data Collected For Group Two

(Natural-language and RTSA Specifications)

Subject	Test Statements total (comments)	Creation Time	Execution Time real (cpu)	Coverage percent (bbs)	Date Received
1	WITHDREW	-----------------	--------------------	-----------------	10/2
2	200 (0)	6.5 h	0.5 s (0.1 s)	87.93 (102)	11/2
3	WITHDREW	----------------	--------------------	----------------	11/23
4	130 (51)	3 h	0.4 s (0.1 s)	88.79 (103)	10/24
5	170 (49)	11 h	0.4 s (0.2 s)	91.38 (106)	10/14
6	WITHDREW	-----------------	--------------------	-----------------	11/12
7	384 (110)	25.5 h	0.8 s (0.4 s)	87.07 (101)	10/20
8	136 (65)	8 h	0.3 s (0.1 s)	88.79 (103)	10/24

In group two, subjects 2 and 7 failed to cover at least 103 basic blocks. Over the course of the experiment, subject 2 provided several excuses to explain why the promised test scripts were late. This pattern of behavior suggests that the subject was not motivated to perform the task. In the case of subject 7, significant effort was expended (25 and 1/2 hours), and yet, only 101 basic blocks were covered. In fact, the test cases were submitted at the tail end of a formal document that: 1) outlined the test plan, 2) presented a schedule for completion (which called for 26.5 hours), 3) raised a number of testing issues, and 4) provided a traceability graph between the state transition diagram and the test cases. After reading the document submitted with the test cases, the author concluded that the subject spent a great deal of time analyzing the cruise control application for ways to improve its utility (clearly outside the scope of the assignment). Judging from the time spent, the document produced, and the coverage achieved, this subject invested a great deal of effort in form at the cost of some substance.

Table 4. Data Collected For Group Three

(Natural-language, RTSA, and Executable Specifications)

Subject	Test Statements total (comments)	Creation Time	Execution Time real (cpu)	Coverage percent (bbs)	Date Received
1	177 (91)	6 h	0.3 s (0.1 s)	92.24 (107)	10/22
2	WITHDREW	------------------	---------------------	------------------	10/15
3	WITHDREW	----------------	---------------------	-----------------	11/23
4	502 (192)	20.5 h	0.6 s (0.2 s)	93.10 (108)	10/29
5	205 (74)	3.5 h	0.3 s (0.1 s)	83.62 (97)	11/5
6	103 (28)	4 h	0.4 s (0.l s)	90.52 (105)	11/2
7	178 (59)	11 h	0.3 s (0.1 s)	84.48 (98)	10/24
8	144 (48)	4.5 h	0.3 s (0.0 s)	83.62 (97)	10/26

In group three, subjects 5, 7 and 8 failed to cover even 100 basic blocks. After discussions with subject 5, the author concluded that the subject had insufficient time to dedicate to the task, but still wished to fulfill the commitment. This led the subject to devote a fixed amount of time to the task, rather than to aim for thoroughness. Further investigation into the case of subject 7 revealed that many of the test cases created were aimed at testing the test language interpreter, not the cruise control software. Apparently, subject 7 misunderstood the assignment. In the case of subject 8, many of the test cases probed around the boundary conditions of vehicle speed. It appears that this subject invested too much time at speed boundaries, sacrificing many cases in the state transition diagram.

Using the test scripts received, some commonly computed statistics were calculated for each group and are displayed in Table 5. Two statistics were used to search for significant differences in performance among the groups. A T-statistic, computed for a non-parametric, rank-sum test, [Spurr and Bonini 1973, pp. 320-324] provided the primary means to identify substantial variation among the groups. To verify these results, for each measure, a standard error test [Spurr and Bonini 1973, pp. 268-273] was used on the difference between the means for each group. Table 6 displays the computed statistics.

Table 5. Some Statistics Regarding The Groups

	Coverage		Creation Time		Real Execution Time	
Group	Mean	Std. Dev.	Mean	Std. Dev.	Mean	Std. Dev.
1	101.63 bbs	4.87 bbs	4.84 h	2.56 h	29.13 s[1]	58.16 s
2	103.00 bbs	1.87 bbs	10.80 h	8.71 h	0.48 s	0.19 s
3	102.00 bbs	5.22 bbs	8.25 h	6.59 h	0.37 s	0.12 s

[1] This high mean resulted from the two subjects who generated automatically a large number of test cases. These test cases took minutes of CPU time to execute. The time taken by most of the test cases was insignificant and, so, execution time is eliminated from further consideration.

Test Coverage Null Hypothesis: No significant difference in test coverage was achieved by any group. The statistical tests on the collected data support the null hypothesis. To identify a significant difference at the 95% confidence level in test coverage between groups 1 and 2 the T-statistic must exceed 70 (74 at the 99% confidence level) or fall below 42 (38 at the 99% confidence level). The corresponding ranges for groups 1 and 3 are 76-44 (80-40 at the 99% confidence level), and for groups 2 and 3 are 42-18 (44-16 at the 99% confidence level). The computed T-statistics fall within the acceptable range. The T-statistics are corroborated by the estimation of standard error in the difference between the mean test coverage of each group. To be significant, at the 95% confidence level, the standard error values must exceed 1.96.

Table 6. Statistics To Test Hypotheses

	Test Coverage		Creation Time	
	T-statistic	Standard Error	T-statistic	Standard Error
Group 1 vs. Group 2	55	0.72	42	1.49
Group 1 vs. Group 3	62.5	0.14	55	1.2
Group 2 vs. Group 3	29	0.44	33.5	0.54

Test Creation Time Null Hypothesis: No significant difference in test creation time exists between any of the groups. The statistical tests on the collected data support the null hypothesis. To the degree that variability is seen in test creation times, the evidence suggests that subjects using the natural-language description alone took less time to generate tests than did subjects in the other two groups.

4.1 Other Possible Explanations

Since both hypotheses were rejected by the experimental evidence, the author considered other possible explanations. An earlier experiment by Curtis, *et al.*, found that for three types of programming tasks (comprehension, modification, and debugging) a link existed between the number of programming languages known by a subject and performance [Curtis et al 1989]. To investigate whether programming language familiarity might explain the differences in individual performance among subjects in the current study, the author queried those subjects who completed the experiment to determine how many programming languages each subject knew well enough to create working programs. This new information, along with some other data already collected, was used to reassess the test coverage achieved by each subject. Table 7 contains the data used for the reassessment; the factors considered include: 1) number of programming languages each subject can use effectively, 2) the amount of time each subject spent creating the tests, 3) the number of years of programming experience for each subject, and 4) the number of years of functional testing experience for each subject. For each of these traits, subjects were formed into two groups and rank-sum tests were used to evaluate the test coverage achieved by each group. The resulting T-statistics are shown in Table 8.

Programming Languages. The subjects shown in Table 7 were divided into two groups: 1) those who knew four or fewer programming languages and 2) those who knew more that four programming languages. A rank-sum T-statistic, yielding 99, was then computed for the first group. This result showed no significant difference in test coverage between the two groups. (To be significant within the 95% confidence interval, the T-statistic must fall outside of the range 72-118; for the 99% confidence interval the T-statistic must fall outside 66-124.)

Programming Experience. The subjects in Table 7 were also divided into two groups depending on the number of years of programming experience. Twelve subjects had ten or more years of programming experience and seven had less than ten years. The computed T-statistic was again 99, showing no significant difference in test coverage based on subject experience.

Functional Testing Experience. The subjects were then examined based on the number of years of functional testing experience: Eight subjects had no experience and 11 subjects had at least some experience. The computed T-statistic, 71, indicated with a 95% confidence (range of 72-118) that those subjects without previous functional testing experience achieved somewhat better test coverage than those subjects with some functional testing experience. At the 99% confidence level (range of 66-124), no difference in performance was found. The difference in

Table 7. Data Used To Reassess Test Coverage Performance Among Subjects

Group-Subject	Programming Languages	Programming Experience	Functional Test Experience	Creation Time (h)	Test Coverage (bbs)
1-1	6	14	0	5	105
1-2	4	13	1	4	101
1-3	6	8	0	4	103
1-4	6	22	10	5	102
1-5	2	7	4	2.75	90
1-6	3	6	3	8	104
1-7	4	13	13	5	104
1-8	3	24	10	5	104
2-2	6	12	3	6.5	102
2-4	3	8	0	3	103
2-5	3	20	3	11	106
2-7	11	17	5	25.5	101
2-8	4	9	0	8	103
3-1	1	10	2	6	107
3-4	8	15	0	20.5	108
3-5	7	7	3	3.5	97
3-6	4	20	10	4	105
3-7	4	25	0	11	98
3-8	8	9	0	4.5	97

Table 8. T-statistics For Various Subject Groupings

Programming Languages (Number) <= 4 vs. > 4	Programming Experience (Years) >= 10 vs. < 10	Functional Testing Experience (Years) 0 vs. > 0	Creation Time (h) <= 5 vs. > 5
99	99	71	123

performance in these two groups was not so great that any strong conclusion can be drawn. Perhaps the subjects without functional testing performance were more careful when creating their tests.

Creation Time. The 11 subjects who spent five hours or less on the task were compared against the eight subjects who spent more than five hours. The resulting T-statistic, 123, was significant at the 95% confidence level (range of 72-118) and almost significant at the 99% confidence level (range of 66-124). This result suggests that those subjects who spent more time creating the tests also achieved the best coverage. This result, though expected, is not very strong.

Since the evidence collected shows no significant difference in performance among the groups in either test coverage achieved or in test creation time, the individual differences among the subjects must be attributed to factors other than the set of specifications provided to the tester, the number of programming languages known, or the number of years of programming and functional testing experience. Several factors appear to account for such individual differences in performance; these factors include: amount of time spent on the task, differences in ability, differences in motivation, and variations in the degree of understanding of the assignment.

5.0 Conclusions

The experiment reported in this paper found that augmenting natural-language descriptions with graphical and executable specifications made no significant improvement in the test coverage achieved by subjects writing functional test cases. Differences in performance were observed among individual subjects. The observed differences were shown not to result from differences in programming experience among the subjects. Some statistical evidence suggested that those subjects without previous functional testing experience performed better than those subjects who had previous functional testing experience. Statistical evidence also suggested that those subjects who spent more time on the task achieved better performance. Some non-statistical evidence hints that subjects who achieved better performance might possess a better understanding of the task or might be more highly motivated to do well on the assignment. Of course, differences in ability among the subjects might also account for the observed differences in performance.

Two flaws in the experiment design, taken together, tended to bias the statistical results against the hypotheses. The test language provided to the participants could be used to achieve a maximum of 94% statement coverage for the system being tested. In addition, the participants were provided with sample tests that achieved 89% statement coverage. These facts limited the possibility of improvement (assuming each participant began with the sample tests) to five percent. For reasons explained in the body of the paper, these flaws did not appear to affect the outcome; however, since the hypotheses were not supported by the statistical results, the experiment should be rerun, after removing the flaws, to verify the results.

Some non-statistical evidence from this experiment suggests that improved functional testing performance might be achieved through attention to management issues: 1) ensure that the task is well-understood, 2) foster motivation among workers, 3) allow enough time for the task, and 4) assign able workers to the job. Statistical experiments should be designed to investigate these possibilities. One can envision an experiment where functional testing tasks are assigned to groups of subjects who have been given varying degrees of instruction regarding the tasks. Another experiment might provide different rewards to groups of subjects who achieve pre-established performance goals. A third experiment might assign functional testing tasks to groups that are each given different amounts of time to complete the task. A fourth experiment might prescreen subjects using some ability test, group the subjects by their performance on that test, and then assess the performance of the grouped subjects given the same functional testing task.

More philosophically, this experiment provides a reminder that software engineering is a business performed by people, for people. New tools and techniques can improve our ability to perform the work of software engineering, but variations in performance can still be expected because the tools and techniques are used by individuals who differ in ability, who vary in their understanding of the job, and who respond differently to different situations. Software engineering researchers should not forget to investigate the people-related, management issues that, perhaps, provide the most significant, short-run influences on the performance of software engineers.

6.0 Acknowledgments

The author extends his thanks to the twenty-four subjects from the Computer Systems Laboratory of the National Institute of Standards and Technology for the time they spent on this experiment; most especially appreciated, of course, are those twenty who completed test scripts. In addition, the author thanks Dr. A. Jefferson Offutt for his insightful comments on early drafts of this paper. The author also benefited from the suggestions, criticisms, and encouragement of his fellow doctoral students.

7.0 References

[Bailin 1989] S. C. Bailin, "An Object-Oriented Requirements Specification Method," *Communications of the ACM*, Volume 32, Number 5 May 1989, pp. 608-623.

[Basili et al 1986] V. Basili, R.W. Selby, and D.H. Hutchens, "Experimentation in Software Engineering," *IEEE Transactions On Software Engineering*, Vol. SE-12, No. 7, July 1986, pp. 733-743.

[Curtis et al 1989] B. Curtis, S. B. Sheppard, E. Kruesi-Bailey, J. Bailey, and D. A. Boehm-Davis, "Experimental Evaluation of Software Documentation Formats", *The Journal of Systems and Software*, Vol. 9, 1989, pp. 167-207. (0164-1212/89)

[Diller 1990] A. Diller, Z An Introduction To Formal Methods, John Wiley & Sons, Chichester, United Kingdom, 1990.

[Gomaa 1993] H. Gomaa, Software Design Methods for Concurrent and Real-Time Systems, Addison-Wesley, Reading, Massachusetts, 1993.

[Harel et al 1987] D. Harel, A. Pnueli, J.P. Schmidt, and R. Sherman, "On the Formal Semantics of State-charts," IEEE Press, 1987, pp. 54-64. (CH2464-6/87/0000/0054)

[Harel et al 1990] D. Harel, H. Lachover, A. Naamad, A. Pneuli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems", *IEEE Transactions On Software Engineering,* Vol. 16, No. 4, April 1990, pp. 403-413.

[Heninger 1980] K. L. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Application," *IEEE Transactions On Software Engineering*, Vol. SE-6, No. 1, January 1980, pp. 2-13.

[ISO 1987] LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behavior, International Standards Organization, ISO DIS 8807, 1987.

[ISO 1989] Estelle Tutorial, International Standards Organization, ISO 9074:1989/Ammendment 1 Annex D.

[Kelly and Murphy 1990] J.P.J. Kelly and S.C. Murphy, "Achieving Dependability Throughout the Development Process: A Distributed Software Experiment," *IEEE Transactions On Software Engineering,* Vol. 16, No. 2, February 1990, pp. 153-165.

[Lee and Sluzier 1991] S. Lee and S. Sluzier, "An Executable Language For Modeling Simple Behavior," *IEEE Transactions On Software Engineering*, Vol. 17, No. 6, June 1991, pp. 527-543.

[Mills 1993] K.L. Mills, Experiment Notebook For An Inft 821 Functional Testing Experiment Involving An Automated Cruise Control System, October 1993.

[NASA 1993] C Language Integrated Production System (CLIPS) Version 6.0 Reference Manual, Volume I: Basic Programming Guide, National Aeronautics and Space Administration, June 1993.

[Spurr and Bonini 1973] W.A. Spurr and C.P. Bonini, Statistical Analysis For Business Decisions, Richard D. Irwin Inc., Homewood, Illinois, 1973.

[Ward and Mellor 1985-1986] P. T. Ward and S. J. Mellor, Structured Development for Real-Time Systems, Yourdon Press, New York, Three Volumes, 1985-1986.

[Zave and Schell 1986] P. Zave and W. Schell, "Salient Features of an Executable Specification Language and Its Environment," *IEEE Transactions On Software Engineering,* Vol. SE-12, No. 2, February 1986, pp. 312-325.